POLUS: A POwerful Live Updating System

Haibo Chen, Jie Yu, Rong Chen,Binyu Zang
Parallel Processing Institute
Fudan University
Shanghai, 200433, China
{hbchen,yujie,chenrong,byzang}@fudan.edu.cn

Pen-Chung Yew
Dept. of Computer Science and Engineering
University of Minnesota at Twin-Cities
Minneapolis, MN 55455, U.S.A.
yew@cs.umn.edu

Abstract

This paper presents POLUS, a software maintenance tool capable of iteratively evolving running software into newer versions. POLUS's primary goal is to increase the dependability of contemporary server software, which is frequently disrupted either by external attacks or by scheduled upgrades. To render POLUS both practical and powerful, we design and implement POLUS aiming to retain backward binary compatibility, support for multithreaded software and recover already tainted state of running software, yet with good usability and very low runtime overhead. To demonstrate the applicability of POLUS, we report our experience in using POLUS to dynamically update three prevalent server applications: vsftpd, sshd and apache HTTP server. Performance measurements show that POLUS incurs negligible runtime overhead: a less than 1% performance degradation (but 5% for one case). The time to apply an update is also minimal.

1. Introduction

The scale of software has increased dramatically in the past two decades, so do the bugs and security vulnerabilities. Despite progress made in software engineering with better programming support, improved developing models and more effective testing tools, it is undeniable that software is still far from perfect, and this trend is likely to continue. Consequently, there has been an increasing number of software updates to fix bugs, close vulnerabilities and evolve with new features.

Unfortunately, traditional software updating approaches usually require stopping the running software, applying the updates and restarting the software again. Such stop-and-restart approaches inevitably disrupt the execution of running services, thus decrease the availability of software. For example, one previous study [12] indicated that 75% of about 6000 outages in highly available applications were caused by hardware and software maintenance. Since such absence-of-service is ill affordable for many mission-critical systems, such as air control systems, credit card authorization and brokerage operations[17], these systems demand highly dependable services and require services to be available 24X7.

Dynamic updating [6, 9, 10, 3, 14], or live updating is a promising software maintenance technique aiming to remedy such situations, yet still much cheaper and less complex compared to hardware based approaches such as hot/cold standby [18]. By allowing the running systems to be updated on-the-fly without service disruption, such a technique has gained considerable interests and popularity from both researchers and practitioners. Nevertheless, there are few dynamic updating systems that are powerful enough to support rich semantics in modern complex applications. For example, few of them could support updates to broadly used multi-threaded systems when changes involve data. Further, to the best of our knowledge, there is still no effective mechanism to roll back already committed updates and to fix already tainted state for currently running software.

This paper presents POLUS, a **PO**werful **L**ive **U**pdating **S**ystem for existing software. POLUS is designed to support realistic software changes involving both code and data. We design POLUS with an attempt to meet the criteria that we believe are required in dynamically updating software nowadays:

1. **Binary Compatibility:** Dynamic updating systems should ensure backward binary compatibility, thus supporting updates to existing binaries and already running software on-the-fly.
2. **Multithreading Support:** As nowadays non-stop software is often implemented using multithreaded programming models, dynamic updating systems should be able to support the prevalent multithreaded software.
3. **Recovery of Tainted State:** To the best of our knowledge, most existing dynamic updating systems inad-

vertently assume the integrity of the targeted software, without any precaution that the targeted software might have already entered a tainted state, such as a deadlock. Therefore, we feel that it is desirable for dynamic updating systems to be able to recover potential tainted states.

4. **Usability and Manageability:** Dynamic updating systems should be simple to use. It should not be difficult to generate patches to feed such systems. Operators should have control over the process of dynamic updates so that one update will not interfere with another. Moreover, an operator should be able to roll back a committed update if it is found to be buggy.

5. **Low Overhead:** Dynamic updating systems should have minimal impact on software during normal execution.

To demonstrate the applicability of POLUS, we have implemented a prototype system that tries to satisfy the desirable criteria above. We evaluate POLUS using three prevalent sever applications that demand non-stop features: *vsftpd* (a commonly used FTP daemon), the *sshd* (secure shell daemon) in OpenSSH suite, and the *apache* HTTP server with multithreading enabled. All updates to these applications are generated from realistic software releases over a relative long period. Although a complete automation of patch code generation is impossible, we have developed tools to generate most parts of the patch code for dynamic update. The performance measurements on these systems show that our approach only incurs a less than 1% performance degradation (but 5% for one case). Also, the time to completely evolve an application into a newer version is minimal.

In summary, our main contributions in this paper are as follows:

1. We design and implement a powerful dynamic software updating system with a rich set of desirable features. To the best of our knowledge, most of these features are absent in other similar systems.

2. We demonstrate that POLUS can deliver realistic updates to real, large and complex server software without disrupting its service. Our experience shows that dynamic update is a promising approach to evolve contemporary complex software.

The next section provides a brief overview of the capabilities of POLUS in terms of the criteria we described above. An outline of the rest of the paper will also be presented.

2. POLUS: Overview and Approach

In this section, we will first provide a brief survey of existing approaches and describe how they fail to meet the desirable criteria. Then, we will provide an overview of our approach in meeting all these criteria. Finally, we will give an overview on the work flow of POLUS.

2.1. Limitations of Existing Approaches

Unfortunately, no existing approach has met all of the desirable criteria. Generally, many existing approaches are *update-point* based [6, 9, 10, 3, 14]. That is, updates to a running system can only be applied at some specific points of execution, e.g., when the code and data to be updated are not being executed or referenced. Otherwise, the system will result in an inconsistent state.

Update-point based approach has several drawbacks in satisfying the mentioned criteria. First, the ability to find a safe *update point* relies heavily on the analysis ability of compilers or programmers. Unfortunately, for flexible languages such as C, compilers often have great difficulties in pointer analysis and alias analysis. Therefore, they have to make conservative assumptions and incur possible false positives in the analysis result. Although there is a recent proposal that facilitates compiler transformations to make programs updatable [14], their approach is only for single-threaded applications, and it does not maintain binary compatibility. Thus, it cannot be applied to compiled binaries and currently running software.

Second, it is difficult to find update points for multi-threaded software, and some modules in a busy system may not even have a safe point [2, 4]. To the best of our knowledge, there is no updatability analysis that can account for multithreaded software to date when changes involve data. If an update point cannot be reached or detected in time, some security updates will be delayed, exposing the vulnerable system to possible attacks.

Finally, for some *update-point* based dynamic updating systems, an operator may have no control over the process of dynamic update, as the time when the system will reach a *safe point* to apply the update is not known to the operator. If an operator has no knowledge of whether the system has completed its current update, another update could be inadvertently applied when an existing update is still in progress.

2.2. Our Approaches

Being aware of the difficulties in *update-point* based approaches, we use a different approach that allows an update to be applied at any time. Our key idea is to allow the coexistence of both the old and the new versions of data, and

maintain the coherence by calling some state synchronization functions whenever there is a write access to either version of the data to be updated: POLUS write-protects either version of the data during the updating process using the debugging APIs provided by operating systems (e.g. *ptrace* in Unix and *DebugActiveProcess* in Windows.). Such APIs allow a process to gain control over another process, and track a write access to the protected data using signal mechanism (catching and checking the *SIGSEGV* signal). When there is no function manipulating the old version of data, the update process can be safely terminated.

In the rest of the section, we will give an overview on how the desirable criteria are satisfied by POLUS:

1. **Binary Compatibility:** Instead of using program transformation or reconstruction to make a program updatable [14], POLUS utilizes the debugging API to gain control over the patching process and modify the state of running program, similar to the approach used in [1]. In addition, not relying on update points eliminates many constraints on the types of admissible updates, and increases the flexibility of updates.

2. **Multithreading Support:** The difficulty in dealing with multithreaded software lies in the fact that there may be several threads concurrently accessing the to-be-updated data. As mentioned earlier, we discard the update-point based approach and instead allow an update to be immediately applied. POLUS will track the write attempts to either version of data and maintain their consistency using state synchronization functions to synchronize the states of the old and the new data.

3. **Recovery of Tainted State:** In our experience, we found that some running software may be already in a tainted state due to internal software bugs or external attacks against known vulnerabilities. Therefore, updating such software without being aware of such situations may cause a system to fail. Although completely solving these problems may be impossible because sometimes it is impossible to know the correct running state, we try to change the software state to some known safe state. Thus, POLUS provides mechanisms in dynamic patches to check for a tainted state and fix it using the provided recovery code if the system is already tainted.

4. **Usability and Manageability:** To ease the burden of operators, we developed a user interface to facilitate the process of updates. Operators only need to tell the system minimal information (process IDs and patch names) to apply an update. The patch process is also visible to operators. Moreover, POLUS allows operators to rollback committed updates.

   To help a user to construct a dynamic patch for POLUS, we provide a source to source compiler that could identify semantic differences between an old version and a new version of source code. Most POLUS patches can be automatically generated, with some occasional manual adjustments.

5. **Low Overhead:** As we use binary rewriting to direct a function call from its old version to a new version, there may be a little overhead due to the function indirection when the software is being evolved into the new version. Indeed, such overhead is very minimal and our performance measurement shows that it is less than 1% for most applications.

2.3. An Overview of POLUS

As shown in Figure 1, POLUS is composed of three components: **a patch constructor**, in the form of a source to source compiler, which detects the semantic differences between two successive software versions and generates the POLUS patch files. **A patch injector**, which is a running process that applies the updates. **A runtime library**, which provides some utility functions to manage POLUS patches for the patch injector.

Figure 1 also shows the life-cycle of software and general work flow of dynamic updating using POLUS. Traditional ways of software evolution involve stopping the running software, applying the updates and restarting the software again, while dynamic updating supports changes to code and data on-the-fly. To retain binary compatibility, a dynamic update to the software can be started in any running version. The static patch is obtained by analyzing the semantic difference of two successive software versions. To facilitate iterative updates, a version file is used to control the renaming of functions and data in the patches. The static patch is then compiled using regular compilers to generate a dynamic patch as a shared library. The POLUS runtime library will be injected into the running software before the first update. The dynamic patch will be injected by the patch injector on-the-fly, facilitated by POLUS runtime library.

In the next four sections, we describe the key issues in design and implementation of POLUS. We first describe the key issues to support dynamic update (section 3). Then, we present an overview of patches in POLUS (section 4.1), and describe in detail the process of patch generation (section 4.2). Later, we describe the mechanism of the patch injector and the shared library (section 5). Finally, we describe the implementation details of POLUS for Linux on x86 platforms.

In addition, we discuss our experience in using POLUS to update three real-life server applications and their relative performance in section 7. Section 8 presents a discussion

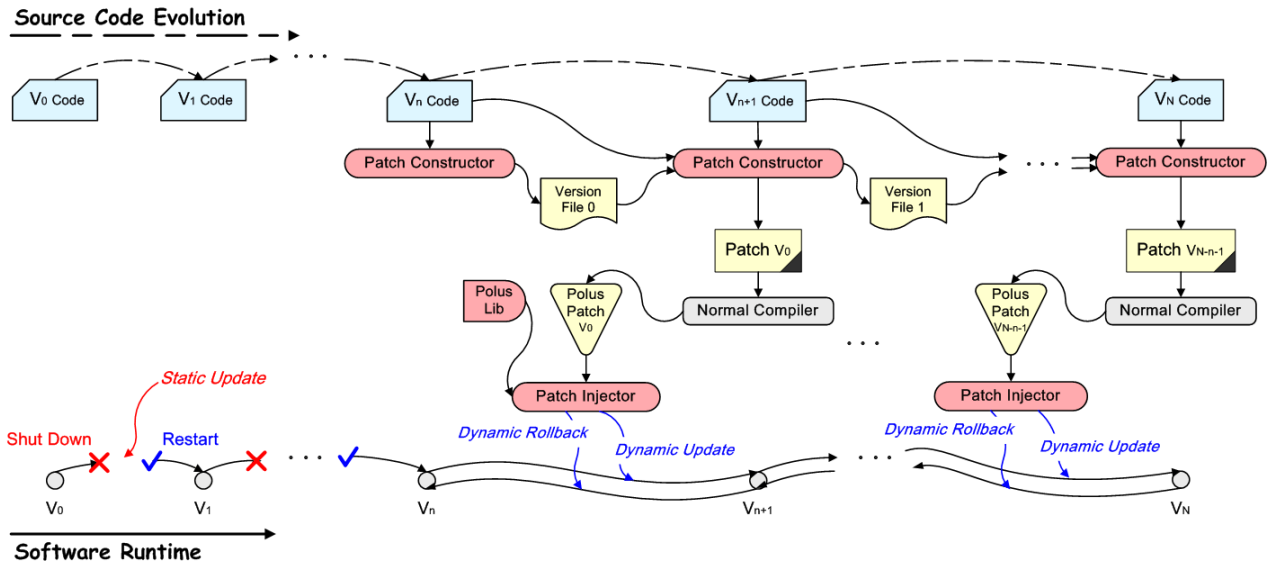

Figure 1. An overview of POLUS and its working flow.

on related work. We close this paper with a discussion on further work and a conclusion.

3. Supporting Dynamic Updates

3.1. Version Management

To support iterative patching, POLUS uses version files [14] to record the patch history of functions, types and variables to avoid naming conflicts. POLUS renames each function, type and variable in the patch file according to its patch history. For example, if function *foo* has been updated three times, then the version number for *foo* in version data file is 3 and its name in the new patch file is *foo_v4*. POLUS maintains a global version to record the total update times. The version for each element in the version file may not be the same if the history of individual updates differs.

3.2. Function Indirection

To implement function indirection, POLUS inserts an indirect *jump* instruction in the prologue of the original function to force all function calls from the old function to the new function.

As POLUS supports iterative updates to a single function, it should be carefully designed to avoid multiple indirections, which can degrade performance. Also, new functions are permitted to directly call other new functions without indirection. As shown in Figure 2, POLUS is carefully designed with a consideration of such cases. When a function is updated, all function indirections from previous versions will directly jump to the newest version. Therefore, the indirection depth will always be no more than one.

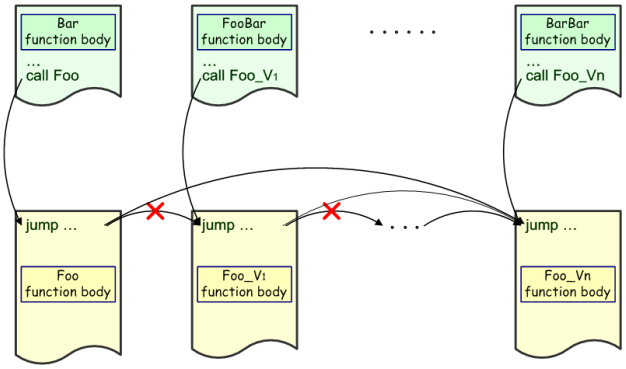

Figure 2. Function Indirection in POLUS.

Keeping all these functions in memory does incur some memory overhead. However, as one of our goals is to support rollbacks of committed patches, keeping them in memory will make rolling back and updating forward more easily.

3.3. State Management

POLUS is a coarse-grained, function-level updating system, that is, it does not take into consideration the local function state (such as local stacks, local variables). Instead, POLUS only considers a global visible state (e.g., global variables) and treats each function manipulating the state as a black box. POLUS is designed to support both single-threaded and multi-threaded applications.

In our approach, both the old and the new instances of data are allowed to co-exist simultaneously. As old functions manipulating the old instance may still be active, there might be concurrent accesses to the old or the new instances. As there should be only one instance of data being active, POLUS must ensure the coherence between them. POLUS employs state synchronization functions provided by the *patch constructor* to maintain the coherence between each pair of instances.

When a dynamic update is being applied, the *patch injector* write protects both the old and the new versions of an instance and associates a signal handler to catch each write attempt to either version of the instance. The signal handler will invoke the corresponding state synchronization functions to transfer the modified state from one version to the other.

In our experience, some of the changed global variables may be read-only throughout their whole life-cycle. For such changes, there is no need to write protect any instance of data and maintain their consistency. As the patch constructor ensures the old (new) instances will only be used by old (new) functions, the old instances will not be used when all old functions accessing them have completed.

3.4. Recovering Tainted States

Existing approaches assume the correctness of the state for running software when an update is being applied. However, it's likely that the running software is buggy and a bug may have occurred. The software may have entered a tainted state (such as a deadlock situation). For example, a known vulnerability on SSL connection in Apache 2.0 will cause a child process to enter an infinite loop, risking denial of service [1]. As a considerable number of software updates are to fix existing bugs, we feel it is necessary to take into consideration the detecting and fixing of the buggy situation.

To support recovery from a tainted state, there are five opportunities in the update process to check and fix such a situation, in the form of callback functions:

1. *pre-update callbacks*, to be called before an update process is started.
2. *thread callbacks*, to be invoked each time a thread leaves a function being updated.
3. *function callbacks*, to be called when all threads have left a function being updated.
4. *data callbacks*, to be invoked when all threads using a data structure have returned from the functions that manipulate the instance of the data structure.
5. *post-update callbacks*, to be called when an update process is to be terminated.

Patch vendors can selectively provide their checking code in these callbacks to detect possible buggy situations and fix them if needed. These callbacks give POLUS opportunities to recover a system from a tainted state. To handle the case for Apache 2.0, one should provide code in the *pre-update callbacks* to check for the infinite loop and break it if necessary. However, not all buggy situations can be easily resolved. For example, in some memory-leaking programs, it will be hard to reclaim all leaked memory if it cannot trace all of them.

3.5. Rolling Back Committed Updates

Some operators choose not to install certain updates due to their lack of confidence in those updates [1]. It is possible that an update might bring new vulnerabilities to the system. In such situations, operators may want to discard some already committed updates. POLUS is designed to provide system administrators with such flexibility.

POLUS treats rollbacks as a special type of updates using existing versions of code and data to update the committed ones. POLUS is carefully designed so that the original code and data could be reused in the rollback process. POLUS use a flag to indicate whether an update is a rollback or a normal update. To support fast rollbacks, POLUS keeps all old versions of code and data in memory. Although it does incur some resource overhead, doing so allows the running software to freely switch among selected versions.

4. POLUS Patches

4.1. An Overview of Dynamic Patches

The notion of POLUS patches are whole-program patches similar to the patches in [9, 14], and are opposed to one patch file per code change used in [10]. We choose one patch file for the whole program because it is easier to ensure the system consistency during the patch process. POLUS patches are coarse-grained, that is, they do not take into consideration the local function state (local variables). Instead, POLUS patches only consider global visible state (such as global variables) and take each function manipulating the state as a black box.

POLUS patches describe the software changes from an old version to a new version. To meet our desirable criteria for a powerful dynamic updating system, POLUS patches should contain rich semantics to express arbitrary changes and fixes to the running software, such as fixing a deadlock situation. Further, as one of the main goals of POLUS is to support contemporary multithreaded software, POLUS

[1] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0748

patches need to contain code that maintains the data consistency among threads when they manipulate shared data structures.

4.2. Patch Construction

An upgrade to software usually involves changes to functions, type definitions and global variables. The main goal of the patch generator is to identify the changes in types, global variables and functions, and the interaction between functions and global variables. This information will be used by the patch injector to apply or rollback POLUS patches.

The first step is the merge process. As POLUS patches are whole-program patches and contemporary software is usually composed of multiple files, we must ensure that if one function or type is changed, all affected code and data must be updated accordingly. For simplicity, POLUS merges all related files into a single file using the merge feature in CIL [15]. In the merge process, POLUS carefully handles the naming conflict by associating clashed names with their filename. To resolve such conflicted names, the patch generator generates code to notify the patch injector to resolve such name conflicts by scanning the original binary file.

Next, the *patch constructor* finds all changed types, global variables and modified functions by comparing the syntax tree of both the old and the new versions of files. For each changed type, the *patch constructor* finds all global variables that derive from the type and adds them to the changed variables list. Then, for each changed variable, a state synchronization function will be generated to maintain the coherence between the old and the new variables. Also, the *patch constructor* gathers all functions that use each changed variable and build the relationship between them. For each modified function, the *patch constructor* detects all changed global variables it uses and builds their relationship.

Finally, the *patch constructor* generates a single file that contains all changed type definitions, all changed global variables, all modified functions, and some support code to build the relationship between variables and functions. This file will be compiled using regular compilers (such as GCC) to generate a dynamic patch file in the form of a shared library.

It's possible that the *patch constructor* may fail to obtain all changed information in the presence of pointer aliases and void pointer casting. To solve this problem, the *patch constructor* will generate warnings that ask operators to adjust the source code or add needed source annotations to make sure that the patches generated are correct.

5. Applying POLUS Patches

Applying a new patch requires three phases. In the first phase, the *patch injector* does some preparation work, such as loading the patch into memory, resolving symbols and registering the patch information to the patch injector. In the second phase, the *patch injector* injects the patch into the running process and maintains the state coherence. Finally, if no thread is executing the old version of modified functions and variables, the patch process can be safely terminated. Rolling back a committed patch is similar except it does not require the preparation work because all required code and data are already in memory.

5.1. Patch Preparation

Two things are done in this phase. First, as mentioned before, POLUS patches are in the form of shared libraries. They must be loaded into memory before being applied. However, there may be some renamed symbols (static variables, renamed symbols due to name clashing) which cannot be resolved using standard linking procedure. The *patch injector* invokes helper functions in *POLUS shared library* to resolve them. As static variables are statically linked, the helper functions scan the original and the new executable files to find addresses for the renamed symbols, using binary utility tools from the compiler tool chain.

Second, as the patch injector must trace the use of changed variables and maintain their consistency, it needs to know the relationship between the functions and variables. By invoking the corresponding code in the POLUS patch, the patch injector will generate the relationship maps describing which variables are used by an updated function and which functions manipulate an updated variable. Also, the patch injector will register the updated functions, data and their corresponding versions. All these patch information will be stored in the *POLUS runtime library* for further use in patch application and rollbacks.

5.2. Patch Application

This phase requires three steps. First, the *pre-update callbacks* will be invoked here if they are provided. The patch injector will get the inflight call graph of each thread in the running software using stack inspection [1]. For a changed variable, if no thread is executing in all functions that reference that variable, then updates to these functions could be done by function indirection and no tracing work is required.

Second, the patch injector write protects all changed global variables currently in use and does the function indirection. Afterwards, all function calls to the original ones will be redirected to the new ones.

Finally, the *patch injector* resumes the execution of the running software. There may be old functions that are still active. They may need to read and write old global variables. To ensure correct execution, the *patch injector* tracks any write access to the new and the old versions of the global variables and synchronizes them by transforming the state from one to the other after the commitment of a write access.

5.3. Patch Termination

The criteria to safely terminate an inflight update are: all threads executing in functions that manipulate changed global variables have been inactive. To decide whether an old function is still active, the *patch injector* maintains a list of active threads for each old function.

At the patch application stage, the list is initialized according to the inflight call graphs. The *patch injector* tracks the thread execution by replacing the return address of the original function with the address of a stub function. The stub function will remove the executing thread from the thread list, invoke the *thread callback*, and return to the caller of the original function. On removing a thread from the thread list, the *patch injector* checks whether the thread list becomes empty or not. If it is empty, the original function is no longer active, and at this time the *function callback* will be invoked. When all functions manipulating a data structure become inactive, the *data callback* will be invoked and the global variable will be marked as unused. When all changed global variables become inactive, the live update process can be safely terminated. The *patch injector* will invoke the *post-update callbacks*, and perform some cleanup work such as restoring the write-protected memory.

6. POLUS Implementation

We have implemented our approach in Linux on x86 platforms. The *patch constructor* is a source-to-source compiler based on CIL-1.3.5 [15] using OCaml. The *runtime library* consists of a number of utility functions to maintain the update information of each software evolution. It is compiled into a shared library to be dynamically linked by each application. We illustrate the detailed implementation of the patch injector by examining the process of loading and applying (or rolling back) an update.

One key issue is how to hijack the running process to be patched (RPP). To apply an update, we run the *patch injector* as a process using *ptrace* to attach the running process to be updated.

POLUS first loads the *POLUS runtime library* and the dynamic patch to RPP's address space. This requires creating a code playground [1] in RPP. More precisely, POLUS maps a range of addresses using *mmap* in RPP, injects code containing *dlopen* and uses *ptrace* to force RPP to execute the injected code. To regain control after the execution leaves the playground, POLUS appends an "*int3*" to any code in the code playground and hijacks the *SIGTRAP* signal.

After the patch has been loaded to RPP, the patch injector invokes code in the patch to retrieve the patch information, and applies the patch. POLUS first replaces the prologue of each affected function with an indirect jump to the new function. As an indirect jump takes up five bytes, POLUS first checks if there is any thread executing in between by iterating the program counter for each thread. POLUS then write protects both the old version and the new version of data structures using *mprotect*. RPP then resumes its normal execution.

To track the write accesses, the *patch injector* hijacks the *SIGSEGV* signal for RPP. On receiving the signal, POLUS unprotects the related data structure and uses *ptrace* to execute the code in a single-step mode (with *PTRACE_SINGLESTEP*). Then, POLUS invokes the provided state synchronization functions to transfer the state from/to the new/old versions of data.

To determine when it's safe to terminate an update, POLUS gains the inflight call graph for each thread by inspecting their call stacks. On inspecting, POLUS replaces the return address of each old function to our supplied stub function. The stub function will remove the calling thread from its active thread list and return to the correct function address. When all functions' thread lists are empty, POLUS restores all write-protected memory and restores RPP to execute in a normal, untraced mode using *ptrace* with *PTRACE_DETACH*. Afterwards, the update process could be safely terminated.

7. Experience and Evaluation

7.1. Experience

To demonstrate the applicability of POLUS, we have used POLUS to dynamically evolve three prevalent long-running server applications into newer versions, over a period of releases:

1. the Very Secure FTP daemon (*vsftpd*), which is the *de facto* FTP server in UNIX environments. We considered the online evolution from 2.0.0 through 2.0.4.

2. the ssh daemon (*sshd*) from the OpenSSH suite, which is a widely-used secure shell daemon. We followed the evolutions from version 3.2.3p1 to 3.6p1.

3. the *apache* HTTP server (*httpd*) , which is a most prevalent HTTP server used nowadays. We tested the upgrades from version 2.1.7 to 2.2.0.

Prog.	First version		Last version		Functions			Types			Global variables		
	Ver.	LOC	Ver.	LOC	Add	Del.	Chg.	Add	Del.	Chg.	Add	Del.	Chg.
vsftpd	2.0.0	13,917	2.0.4	14,293	10	4	81	2	0	16	12	1	3
sshd	3.2.3	54,360	3.6	56,960	32	9	512	1	2	6	27	3	11
httpd	2.1.7	319,366	2.2.0	315,381	18	3	195	4	1	5	12	3	3

Table 1. Update information for three applications over time.

The *vsftpd* and *sshd* are single thread software, while *httpd* is usually configured as multithreaded software.

Table 1 shows the evolution history of the three applications. We report the total number of changes to functions, types and global variables from the starting version to the last updated version. The number of changes is somewhat larger than other approaches due to the fact that POLUS uses function-level updating. For example, if a type is changed, then all affected functions and variables are affected. Nevertheless, we believe it is worthwhile as POLUS retains binary compatibility compared to the compiler-transformation approach [14].

The work flow to upgrade the software is shown in Figure 1. Upgrades can take place in arbitrary running version.

Recovery of Tainted State: POLUS is designed to support recovery from a tainted state. In upgrading *sshd*, we found all versions prior to 3.7.1 contain possible buffer management errors [2]. To detect and resolve such situation, we added checking and recovering code in *pre-update callbacks* in the dynamic patch to check whether the buffer size is valid for each global variable derived from *Buffer* type. Also, we added such code in the *function callback* for each function manipulating such global variables. If the size of a buffer exceeds the defined threshold, the buffer will be truncated in case of a heap overflow.

Although it is sometimes difficult to fix a tainted state and resolving it requires a vulnerability-specific knowledge, we believe our work has raised an important issue to researchers and practitioners about the detection and recovery of tainted states during dynamic updating of running software. Further, to the best of our knowledge, POLUS is the first system to provide mechanisms to resolve such issues.

7.2. Experimental Results

In this section, we present our experimental results using some "real world" tests to get the runtime performance.

The experiments were conducted on a dual Xeon 2.4GHZ server with 2GB RAM, a 1G Realtek 8169 NIC in 100M LAN, and a single 73G 10k RPM SCSI disk. The systems were configured with a Linux Enterprise edition 4, with kernel version 2.6.9-5.ELsmp. The compiler used is gcc-3.4.3 at optimization level -O2. The reported result is a median of 10 runs.

Relative Performance: For the three applications, we measured their performance using two metrics: connection time and transfer rate. The test methodologies are shown in Table 2. To measure the performance of apache, we used *ab* (apache benchmark) to issue 50,000 requests for a single 2.4 KB file, with 500 simultaneous threads.

As shown in Table 3, POLUS incurs undetectable performance overhead. This reflects the fact that the only runtime overhead in POLUS is in function indirection. Although the connection time for *vsftpd* increases about 5%, we don't think such a 0.4 ms difference is noticeable in practice.

Update Time: Generally, the update time of an application is decided by the amount of changed types, variables and affected functions. We measured the update time for applying and rolling back an update. We first updated an application from its first version to its last version and then rolled back to the first version. Figure 3 shows the total update time (Tot.) and the real update time (Real.) of our measured three applications. The total update time includes the whole process to apply a patch. It includes loading the patch into memory, doing necessary preparation work and applying the patch. By contrast, the real update time counts the time to apply a dynamic patch, during which there may be some performance degradations to the running systems.

As shown in Figure 3, the total time to apply/roll back an update is modest and the real update time is rather little. Apache httpd requires relative longer update compared to sshd and vsftpd because it is multithreaded and all its threads needed to be updated. All updates are finished within less than one fifth of a second. The real update time is only several dozens of milliseconds.

Service Disruptions: To measure the impact of dynamic updates on running services. We used *ab* to issue 15,000 requests for a single 2.4 KB file and collected the throughput of apache httpd server when an update from version 2.1.7 to 2.1.8 is in progress. Figure 4 depicts the curve for throughput in the whole process of updating. The time to evolve apache httpd is still very little even under a heavy load. There is only a modest amount degradation (about 30%) during the update. Therefore, we can conclude that POLUS has little impact on running software.

[2] http://www.cert.org/advisories/CA-2003-24.html

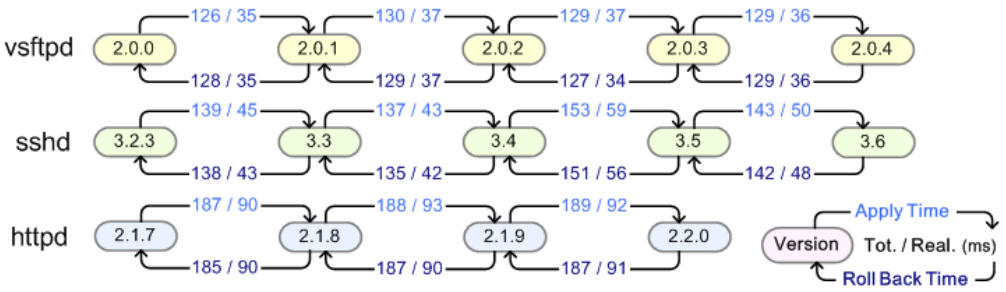

Figure 3. Total update time and real update time for the three applications (in millisecond).

Apps	connection time	transfer rate
vsftpd	average time of requesting 1,000 empty files using *wget*	download rate of a single 222MB file
sshd	total elapsed time of 1,000 requests divided by 1,000	use *scp* to copy a single 138MB file
httpd	use ab (apache benchmark) with "ab -n 50000 -c 500 http://localhost/apache_pb2.gif"	

Table 2. Test methodologies for the three applications.

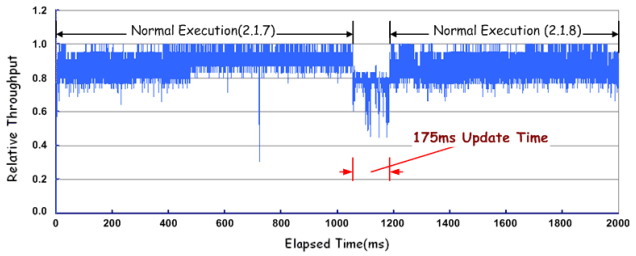

Figure 4. Impact of live update on *httpd* under a heavy load.

8. Related Work

A considerable number of systems have been proposed to dynamically update running software. We compare our approach with some of those systems in terms of the desirable criteria mentioned in section 1.

Ginseng[14] is a recent system for dynamic software updating. By using compiler transformation to make software dynamically updatable, Ginseng made substantial improvements over their previous work [10], which required the program to be designed to be updatable (e.g. specifying when to update). Moreover, Ginseng shows its practicality by applying it to three commodity server applications. In contrast to POLUS, using compiler transformation fails to meet our first goal of binary compatibility, and limits its application to already running software. Further, Ginseng and its predecessor are only applicable to single-threaded software to date. Finally, they lack mechanisms such as rollback support and recovery from tainted states, and their updates can only be applied at specific update points. Nevertheless, Ginseng made a large stride in making dynamic software updating practical to commodity software.

Many other systems lack comparable functionalities or are under rigid restrictions. For example, some systems require the program to be specially constructed [6, 8] in a top-down style of programming, or designed with dynamic updating in mind [11, 10]. Some systems only support changes to abstract data types [5], or do not support changes to interfaces [16, 11]. Some systems only support changes to code [1], or disallow updates to currently active code [7, 6, 8, 13, 2]. None of them satisfies our criteria in terms of backwards binary compatibility, multithreading support, recovery from a tainted state and rollback support, when changes involve both code and data.

LUCOS [4] might be the most similar system. However, LUCOS supports on-the-fly updates to contemporary operating systems, using system virtualization techniques. In our current work, we apply similar concepts to application software, yet without an additional virtualization layer as in LUCOS. Further, to reduce the tedious work in patch construction, we provide compiler support to automate most of the work.

There are some theoretical efforts on the safety of dynamic update. For example, Gupta [9] used formal methods to understand the validity of a dynamic update and proved that finding safe update points to apply updates is, in general, undecidable. Proteus [19] examined the safety of a dynamic update and proposed the notion of representation consistency. However, to the best of our knowledge, their analysis is only applicable to single-threaded, update-point based systems. In POLUS, since both the old and the new versions of code and data are allowed to co-exist, there is no update time-line issue and no need for representation consistency.

Application	connection time (ms)			Transfer rate (MB/s)		
	orig.	upd.once	upd.mult	orig.	upd.once	upd.mult
vsftpd	7.626	7.666	8.015	11.62	11.62	11.63
sshd	146.2	146.6	146.7	11.53	11.54	11.54
httpd	108.4	108.3	108	11.57	11.58	11.50

Table 3. Performance data for original, updated once and updated multiple times applications.

9. Conclusion and Future Work

We have presented POLUS, a powerful live updating system for contemporary server software. In contrast to previous systems, POLUS is capable of updating multithreaded software, and is designed with an awareness of supporting recovering tainted software states and rolling back committed updates, yet with good usability and backwards binary compatibility. Our results suggest that POLUS has negligible impact on application performance. We plan to apply our approach to a wider range of real-life software in the future.

Current implementation of POLUS retains binary compatibility because we feel it necessary to support legacy systems or already running software. Doing so complicates the implementation of POLUS and may make it difficult to handle some infinite loops during update (although updates to the calling functions of such loops are rather rare in practice). For newly developed software, we plan to use compiler transformations (as in [14]) to make the program more friendly to POLUS, thus making the implementation of POLUS easier and reducing some update-time overhead.

10. Acknowledgement and Availability

This work was funded by China National 973 Plan under grant numbered 2005CB321905 and Intel University Research Grant. The source code of POLUS and evolutionary tests are available in sourceforge (`http://sourceforge.net/projects/polus/`). The patch generator is distributed under the BSD License, others are based on GNU General Public License.

References

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Proc. USENIX Security*, Baltimore, MD USA, 2005.

[2] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proc. USENIX ATC*, pages 279–291, Anaheim, CA, USA, April 2005. USENIX Association.

[3] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA*, October 2003.

[4] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *Proc. VEE*, pages 35–44, Ottawa, Canada, June 2006.

[5] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proc. ICSE*, pages 470–476, 1976.

[6] O. Frieder and M. E. Segal. On dynamically updating a computer program: from concept to prototype. *The Journal of System Software*, 14(2):111–128, 1991.

[7] S. Gilmore, D. Kirli, and C.Walton. Dynamic ml without dynamic types. Technical Report ECS-LFCS-97-378, University of Edinburgh, 1997.

[8] D. Gupta. *On-line Software Version Change*. PhD thesis, Indian Institute of Technology, Kanpur, November 1994.

[9] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transaction on Software Engineering*, 22(2):120–131, February 1996.

[10] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proc. PLDI*, pages 13–23, 2001.

[11] G. Hjalmtysson and R. Gray. Dynamic c++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX ATC*, 1998.

[12] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ACM SIGOPS Operating Systems Review*, 38(5):211–223, 2004.

[13] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *Proc. ECOOP*, pages 337–361, 2000.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proc. PLDI*, pages 72–83, June 2006.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. CC'02*, pages 213–228, 2002.

[16] H. M. Orso A., Rao A. A technique for dynamic updating of java software. In *Proc. ICSM*, pages 649–658, 2002.

[17] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC):Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[18] D. Pescovitz. Monsters in a box. *Wired*, 8(12):341–347, 2000.

[19] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proc. POPL*, pages 183–194, January 2005.